Efficient Sparse LU Factorization with Partial Pivoting on Distributed Memory Architectures

Cong Fu and Tao Yang
Department of Computer Science
University of California
Santa Barbara, CA 93106
{cfu,tyang}@cs.ucsb.edu

Abstract

Sparse LU factorization with partial pivoting is important to many scientific applications, but the effective parallelization of this algorithm is still an open problem. The main difficulty is that partial pivoting operations make structures of L and U factors unpredictable beforehand. This paper presents a novel approach called S^* for parallelizing this problem on distributed memory machines. S^* incorporates static symbolic factorization to avoid run-time control overhead and uses nonsymmetric L/U supernode partitioning and amalgamation strategies to maximize the use of BLAS-3 routines. The irregular task parallelism embedded in sparse LU is exploited using graph scheduling and efficient run-time support techniques. The experimental results on the Cray-T3D with a set of Harwell-Boeing nonsymmetric matrices are very encouraging and good scalability has been achieved. Even compared to a highly optimized sequential code, the parallel speedups are still impressive considering the current status of sparse LU research.

Keywords: Sparse LU factorization, Column partial pivoting, Dense structures, Task scheduling, Irregular parallelism, Run-time support, Symbolic factorization.

1 Introduction

Sparse matrix factorization is the key to solving a sparse system of linear equations. If a matrix is symmetric and positive definite, Cholesky factorization can be used, for which fast sequential and parallel algorithms have been developed in [16, 21, 22]. However in many applications such as circuit simulation, computational fluid dynamics and structural mechanics, the associated equation systems involve nonsymmetric matrices. Pivoting must be conducted to maintain numerical stability for such nonsymmetric linear systems and a typical strategy is partial column pivoting [3, 15]. Because the pivoting operations interchange rows based on the numerical values of matrix elements during the elimination process, it is impossible to predict the precise structures of L and U factors without actually performing the numerical factorization. The adaptive and irregular nature of sparse LU data structures makes the efficient implementation of this algorithm very hard even on a modern sequential machine with memory hierarchies.

There are several approaches that can be used for solving nonsymmetric systems. One approach is the unsymmetric-pattern multi-frontal method [17] that uses elimination graphs to model irregular parallelism and guide the parallel computation if the pivoting sequence can be known prior to numerical factorization. Another approach [10] is to restructure a sparse matrix into a bordered block upper triangular form and use a special pivoting technique which preserves the structure and maintains numerical stability at acceptable levels. It is implemented on Illinois Cedar multi-processors based on Aliant shared memory clusters. Our paper focuses on the parallelization issues for a given matrix ordering with a commonly used pivoting strategy (column partial pivoting) to maintain numerical stability. Parallelization of sparse LU with partial column pivoting is also studied in [11, 12] on a shared memory machine. Their approaches overestimate the nonzero fill-ins by using a static symbolic LU factorization so that the dynamic variation of LU data structures is avoided. They have obtained good speedups for up to 6 processors on a Sequent machine and further work is needed to assess the performance of the sequential code.

As far as we know, there is no published result for parallel sparse LU on current commercially available distributed memory machines such as Cray-T3D, Intel Paragon, IBM SP/2, TMC CM-5 and Meiko CS-2. One difficulty in the parallelization of sparse LU on these machines is how to utilize the sophisticated uni-processor architecture. The design of a sequential algorithm must take advantage of caching, which makes some previously proposed techniques less effective. On the other hand, a parallel implementation must utilize the fast communication mechanisms available on these machines. It is easy to get speedups by comparing a parallel code to a sequential code which does not fully exploit the uni-processor capability, but it is not as easy to parallelize a highly optimized sequential code. One such sequential code is SuperLU [4] which uses a supernode approach to conduct sequential sparse LU with column partial pivoting. The supernode partitioning makes it possible to perform most of the numerical updates using BLAS-2 level dense matrix-vector multiplications, and therefore to better exploit memory hierarchies. They perform symbolic factorization and generate supernodes on the fly as the factorization proceeds. Their code delivers impressive performance and is among the best sequential codes for sparse LU with partial pivoting [4, 2]. However it is challenging to parallelize their code to get scalable performance and so far we have not seen any published results on the parallelization of their method on distributed memory machines.

In this paper, we present a novel approach that considers three key optimization strategies together in parallelizing the sparse LU algorithm: 1) adopt a static symbolic factorization scheme to eliminate the data structure variation caused by dynamic pivoting; 2) identify data regularity from the sparse structure obtained by the symbolic factorization scheme so that efficient dense operations can be used to perform most of the computation; 3) make use of graph scheduling techniques and efficient run-time support to exploit irregular parallelism. We observe that on most current commodity processors with memory hierarchies, a highly optimized BLAS-3 subroutine usually outperforms a BLAS-2 subroutine in implementing the same numerical operations [3, 5]. We can afford to introduce some extra BLAS-3 operations in re-designing the LU algorithm so that the new algorithm is easy to be parallelized but the sequential performance of this code is still competitive to the current best sequential code. We use the static symbolic factorization technique first proposed in [11, 12] to predict the worst possible structures of the L and U factors without knowing the actual numerical values, then we develop a nonsymmetric L/U supernode partitioning technique to identify the dense structures in both L and U factors, and maximize the use of BLAS-3 level subroutines (matrix-matrix multiplication) for these dense structures. We also incorporate a supernode amalgamation technique to increase the granularity of the computation.

In modeling the irregular parallelism in the re-designed sparse LU algorithm, we use directed acyclic task graphs (DAGs). Scheduling and executing DAG parallelism is a difficult job because parallelism in sparse problems is irregular and execution must be asynchronous. We need to overlap computation with communication, balance loads and eliminate unnecessary communication overhead. We have examined two possible scheduling strategies. One is to use the compute-ahead scheduling [15] and another is to use sophisticated graph scheduling. We implemented our sparse LU method using a run-time system RAPID [8, 9] and conducted experiments with a set of Harwell-Boeing nonsymmetric matrices on Cray-T3D. Our experiments show that the scheduled code has achieved good scalability.

The rest of the paper is organized as follows. Section 2 gives the problem definition. Section 3 describes how to predict data structures for sparse LU factorization. Section 4 presents a nonsymmetric L/U supernode partitioning. Section 5 describes the program partitioning and the task graph definition. Section 6 addresses the scheduling and run-time support issues. Section 7 discusses a supernode amalgamation technique to adjust task granularity. Section 8 presents the experimental results. Section 9 concludes the paper.

2 Preliminaries

Sparse LU factorization can be used to solve large linear systems arising from many important applications [3, 15, 20]. Given a nonsingular matrix A, it can be factorized into two matrices L and U using Gaussian Elimination with column partial pivoting. During each step of the elimination process, a row interchange may be needed to maintain numerical stability. The elimination process is listed in Figure 1 and the result of the factorization process can be expressed by:

$$PA = LU,$$

where L is a unit lower triangular matrix, U is a upper triangular matrix, and P is a permutation matrix which contains the row interchange information. The solution of a linear system $Ax = b$ can hence be solved by two triangular solves: $Ly = Pb$ and $Ux = y$. The triangular solves are much less time consuming than the Gaussian Elimination process.

When A is a nonsymmetric sparse matrix, the structures of triangular factors L and U depend on not only the sparsity structure of A but also the numerical values of A. Most sparse LU algorithms

```
for k = 1 to n-1
    Find m such that |a_mk| = max_{n>=i>=k} {|a_ik|};
    if a_mk = 0, then A is singular, stop;
    Swap row k with row m;
    for i = k+1 to n with a_ik != 0
        a_ik = a_ik / a_kk;
    endfor
    for j = k+1 to n with a_kj != 0
        for i = k+1 to n with a_ik != 0
            a_ij = a_ij - a_ik * a_kj;
        endfor
    endfor
endfor
```

Figure 1: Gaussian elimination with column partial pivoting for LU factorization.

involve dynamic memory allocation during the factorization process. For example, the SuperLU package [4] first makes a rough estimation on the total storage needed, or more precisely according to the results from previous factorization, and expands the storage when it becomes necessary later during the factorization.

Caching behavior plays an important role in achieving good performance for scientific computations. To better exploit memory hierarchy in modern architectures, supernode partitioning is an important technique to exploit the regularity of sparse matrix computations and utilize BLAS operations to take advantage of dense operations. It has been successfully applied to Cholesky factorization [18, 21, 22]. The difficulty for the nonsymmetric case is that supernode structure depends on pivoting choices during the factorization and thus cannot be determined in advance. The SuperLU performs symbolic factorization and identifies supernodes on the fly. It also maximizes the use of BLAS-2 level operations to improve the cache performance of sparse LU. As a result, it outperforms several other sparse LU packages [4]. For instance, it is better than UMFPACK [2] for 15 out of 21 testing matrices.

It is challenging to parallelize SuperLU and get good speedups because of the following reasons.

- One has to parallelize the symbolic factorization part. Otherwise the speedups will be limited by the symbolic factorization which contributes $20-45\%$ of the total execution time for most of the tested matrices [4].

- Even if the symbolic factorization is parallelizable, there exists another difficulty for parallelization, especially on distributed memory machines. The pivot selection and row interchange dynamically increase fill-ins, which in turn will cause changes of L/U data structures, data dependence structures and processor loads. Using the precise pivoting information at each elimination step can certainly optimize data space usage, reduce communication and improve load balance, but such benefits could be offset by high run-time control overhead.

The strategy of static data structure prediction in [12] is valuable in avoiding dynamic symbolic factorization, identifying the maximal data dependence patterns and minimizing dynamic control overhead. We will use this static strategy. But the overestimation does introduce extra fill-ins and some unnecessary operations in the numerical factorization.

We observe that in SuperLU [4] the `DGEMV` routine (the BLAS-2 level dense matrix vector multiplication) accounts for 78% to 98% of the floating point operations (exclude the symbolic factorization part). It is also a fact [3] that BLAS-3 routine `DGEMM` (matrix-matrix multiplication) is usually much faster than BLAS-1 and BLAS-2 routines. On Cray-T3D with a matrix of size 25×25, `DGEMM` can achieve 103 MFLOPS while `DGEMV` only reaches 85 MFLOPS. Thus the key idea of our approach is that if we could find a way to maximize the use of `DGEMM` after using static symbolic factorization, even with overestimated nonzeros and extra numerical operations, the overall code performance could still be competitive to SuperLU which mainly uses `DGEMV`.

3 Storage prediction for L and U factors

The purpose of the symbolic factorization is to obtain structures of L and U factors. Since pivoting sequences are not known until the numerical factorization, the only way to allocate enough storage space for the fill-ins generated in the numerical factorization phase is to overestimate. Given a sparse matrix A with a zero-free diagonal, a simple solution is to use the Cholesky factor L_c of $A^T A$. It has been shown that the structure of L_c can be used as an upper bound for the structures of L and U factors regardless of the choice of the pivot row at each step [11]. But it turns out that this bound is not very tight. It often substantially overestimates the structures of the L and U factors.

We consider another method instead [11]. The basic idea is to statically consider all the possible pivoting choices at each step. The space is allocated for all the possible nonzeros that would be introduced by *any* pivoting sequence that could occur during the numerical factorization. We summarize the symbolic factorization method briefly as follows.

The nonzero structure of a row is defined as a set of column indices at which nonzeros or fill-ins are present in the given $n \times n$ matrix A. Since the nonzero pattern of each row will change as the factorization proceeds, we use $\mathcal{R}_i^k$ to denote the structure of row i after step k of the factorization and A^k to denote the structure of the matrix A after step k. And a_{ij}^k denotes the element a_{ij} in A^k. Notice that the structures of each row or the whole matrix cover the structures of both L and U factors. In addition, in the process of symbolic factorization we assume that no exact numerical cancellation occurs. Thus, we have

$$\mathcal{R}_i^k = \{j | a_{ij}^k \text{ is structurally nonzero}\}.$$

We also define the set of candidate pivot rows at step k as follows:

$$\mathcal{P}_k = \{i | i \geq k, \text{and } a_{ik}^{k-1} \text{ is structurally nonzero}\}.$$

We assume that a_{kk} is always a nonzero. For any nonsingular matrix which does not have a zero-free diagonal, it is always possible to permute the rows of the matrix so that the permuted matrix has a zero-free diagonal [6]. Though the symbolic factorization does work on a matrix that contains zero entries in the diagonal, it is not preferable because it makes the overestimation too generous. The symbolic factorization process will iterate n steps and at step k, for each row $i \in \mathcal{P}_k$, its structure will be updated as:

$$\mathcal{R}_i^k = \mathcal{R}_i^{k-1} \bigcup \Big(\bigcup_{j \in \mathcal{P}_k, j \neq i} \mathcal{R}_j^{k-1} - \{1, 2, \cdots, k-1\} \Big).$$

Essentially the structure of each candidate pivot row at step k will be replaced by the union of the structures of all the candidate pivot rows except those column indices less than k. In this way it is guaranteed that the resulted structure A^n will be able to accommodate the fill-ins introduced by any possible pivot sequence. A simple example in Figure 2 demonstrates the whole process.

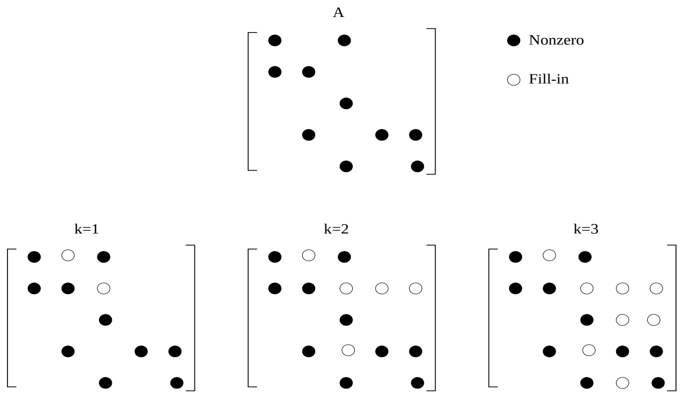

Figure 2: The first 3 steps of the symbolic factorization on a sample 5×5 sparse matrix. The structure remains unchanged at steps 4 and 5.

This symbolic factorization is applied after an ordering is performed on the matrix A to reduce fill-ins. The ordering we are currently using is the multiple minimum degree ordering for $A^T A$. We also permute the rows of the matrix using a transversal obtained from Duff's algorithm [6] to make A have a zero-free diagonal. The transversal can often help reduce fill-ins [7].

In the SuperLU, symbolic factorization is conducted dynamically according to the actual pivoting choice so that overestimation is not an issue. But the symbolic factorization contributes average $20 - 45\%$ to the total factorization time for the tested matrices. Even for matrices with the same sparsity structure but with different numerical values, the dynamic symbolic factorization has to be re-performed. One advantage of the static symbolic factorization is that the result can be reused for matrices with the same initial nonzero structure. A set of nonsymmetric testing matrices from the Harwell-Boeing collection are listed in Table 1. We have compared the number of nonzeros obtained by the static approach and the number of nonzeros obtained by SuperLU for these matrices. The results in Table 2 show that the over-estimation usually leads to about $30 - 50\%$ more nonzeros, which is acceptable.

The extra nonzeros do imply additional computational cost. For example, one has to either check

Matrix	Identifier	Order	Nonzeros
1	sherman5	3312	20793
2	lnsp3937	3937	25407
3	lns3937	3937	25407
4	sherman3	5005	20033
5	jpwh991	991	6027
6	orsreg1	2205	14133
7	saylr4	3564	22316

Table 1: Testing matrices from the Harwell-Boeing collection.

Matrix	1	2	3	4	5	6	7
NZ(Static)	326434	694257	709337	625063	205016	585686	986203
NZ(SuperLU)	250108	453904	459002	443349	141927	414699	670964
Ratio	1.31	1.53	1.55	1.41	1.44	1.41	1.47

Table 2: The number of nonzeros obtained by the static symbolic factorization and the dynamic approach in SuperLU.

if a symbolic nonzero is an actual nonzero during the numerical factorization, or directly perform arithmetic operations which could be unnecessary. If we can aggregate these element-wise floating point operations and maximize the use of BLAS-3 subroutines, the sequential code performance will still be competitive. Thus it is necessary to identify dense structures in a sparse matrix after the static symbolic factorization.

4 L/U supernode partitioning and dense structure identification

Supernode partitioning is a commonly used technique to improve the cache performance of sparse code [1]. For a symmetric sparse matrix, a supernode is defined as a group of contiguous columns that have identical nonzero patterns. Excellent performance has been achieved in [18, 21, 22] using supernode partitioning for Cholesky factorization. However, the above definition is not directly applicable to sparse LU with nonsymmetric matrices. A good analysis for defining unsymmetric supernodes is available in [4]. Basically there could be several types of unsymmetric supernodes, namely T1, T2, T3 and T4 based on different criteria. Their results show that T2 and T3 are the best choices in terms of performance. In this paper we choose type T2 supernode. It is defined as a group of consecutive columns in which the corresponding L factor has a dense lower triangular diagonal block and the same nonzero pattern below the diagonal block. Based on this definition, in each column block the L part only contains dense subrows. We call this partitioning method L supernode partitioning. Here by "subrow" we mean the contiguous part of a row in a supernode. However, the issue of identifying dense structures in a U factor is not addressed by supernode types T2 and T3 [4] and a standard compressed sparse column format is used to store a U factor.

For the dynamic pivoting approach in SuperLU, it is difficult to explore structure regularity in a U factor after L supernode partitioning. However in our approach, there are dense columns (or subcolumns) in a U factor that we can identify after the static symbolic factorization. We discuss the U partitioning strategy as follows. After a L supernode partition has been obtained on a sparse matrix A, i.e., a set of column blocks with possible different block sizes, the same partition is applied to the rows of the matrix to further break each supernode into submatrices. Now each off-diagonal submatrix in the L part is either a dense block or contains dense blocks. Furthermore, we use the following theorem to identify dense subcolumn structures in the U factor. This is the key to maximize the use of BLAS-3 subroutines in our algorithm.

In the following theorem, we show that our strategy will be successful and there is a rich set of dense structures to exploit. The following notations will be used through the rest of the paper.

- The L and U partitioning divides the columns of A into N column blocks and the rows of A into N row blocks[1] so that the whole matrix is divided into N^2 submatrices. For submatrices in the U factor, we denote them as U_{ij} for $1 \leq i < j \leq N$. For submatrices in the L factor, we denote them as L_{ij} for $1 \leq j \leq i \leq N$. If $i = j$, L_{ii} denotes the diagonal submatrix.

- Define $S(i)$ as the starting column (or row) number of the i-th column (or row) block.

- A subcolumn (or subrow) is a column (or row) in a submatrix. For simplicity, we use a global column (or row) index to denote a subcolumn (or subrow) in a submatrix. For example, by subcolumn k in the submatrix block U_{ij}, it means the subcolumn in this submatrix with the global column index k where $S(j) \leq k < S(j+1)$. Similarly we use a_{ij} to indicate an individual nonzero element based on global indices. A *compound structure* in L or U is a submatrix, a subcolumn, or a subrow.

- A compound structure is nonzero if it contains at least one nonzero element or fill-in. In the following we will not differentiate between nonzero and fill-in entries. They are all considered as nonzero elements. Notice that an algorithm only needs to operate on nonzero compound structures. A compound structure is dense if all of its elements are nonzeros.

- Recall that $\mathcal{R}_i^k$ is the nonzero pattern of row i after step k in the symbolic factorization and $\mathcal{P}_k$ is the set of candidate pivot rows at step k.

- Let $T(a_{ij})$ be the time stamp at which the nonzero element a_{ij} in a matrix is created during the symbolic factorization. The time stamp of an original nonzero element is 0. If a nonzero is first created at the k-th step of the symbolic factorization, its time stamp is k.

[1]Notice that N is not necessary the number of supernodes because in order to fit into cache, we have to break big supernodes into smaller ones.

Theorem 1 *Given a sparse matrix A, after the above static symbolic factorization and 2-D non-symmetric L/U supernode partitioning are performed on A, each nonzero submatrix in the U factor of A contains only dense subcolumns.*

Proof: For a nonzero subcolumn m in any nonzero submatrix U_{ij} as shown in Figure 3(a), let a_{km} be the nonzero element in the subcolumn m with the smallest row number k, where $k < m$, $i < j$, $S(i) \leq k < S(i+1)$ and $S(j) \leq m < S(j+1)$. We want to prove that subcolumn m in U_{ij} is dense.

We will use the following two facts through the rest of the proof.

- **Fact 1.** For any $1 \leq s, t \leq N$, $T(a_{st}) \leq s$ and $T(a_{st}) < t$. This is because from any step x, the symbolic factorization only affects the nonzero patterns of the portion of the matrix that starts from row x (include row x) through the last row and from column x (exclude column x) through the last column.

- **Fact 2.** The diagonal submatrix L_{ii} is dense in the lower triangular part after L supernode partitioning. Therefore the lower triangular part of any subcolumn with global index less than or equal to y in L_{ii} must be dense at step y.

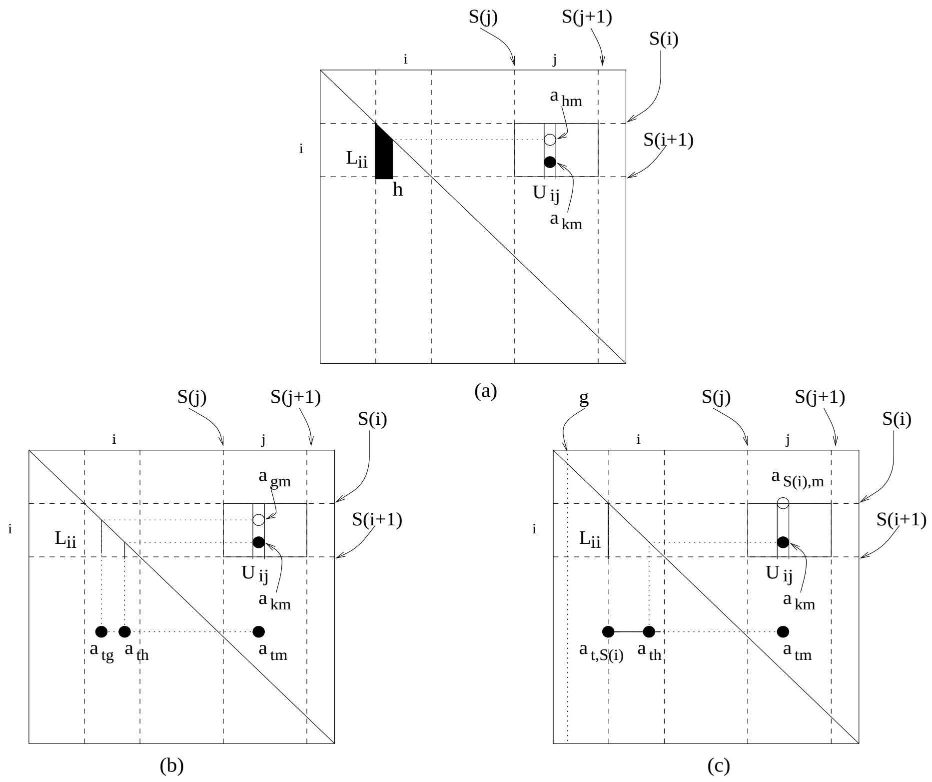

Figure 3: An illustration for the proof of Theorem 1. (a) Situation for Case 1; (b) Situation for Sub-case 2.1; (c) Situation for Sub-case 2.2.

Let $h = T(a_{km})$. Thus $h \leq k$ by Fact 1. If $k = S(i)$, i.e., a_{km} is the first nonzero element of the subcolumn m in the submatrix U_{ij}, then a_{km} must be created before step $S(i)$ or at step $S(i)$ due to the static pivoting method. Thus at step k, the nonzero element a_{km} will lead to a complete fill-in of the subcolumn m in the U_{ij} because all the rows from $S(i)$ to $S(i+1)-1$ belong to $\mathcal{P}_k$ due to Fact 2.

It is impossible that $k > S(i)$ and we prove it by contradiction. Suppose that $k > S(i)$, we need to consider the following two cases. Notice that $h > S(i)$ otherwise a_{km} is created before or at step $S(i)$ and this element will lead to the creation of $a_{S(i),m}$ at step $S(i)$.

Case 1: If $h < k$, i.e., a_{km} is created before step k, then since the lower triangular part of subcolumn h in L_{ii} is dense at step h due to Fact 2, this will lead to the creation of another nonzero element a_{hm} at this step, which contradicts the fact that a_{km} is the first nonzero element of the subcolumn m in U_{ij}. This is illustrated in Figure 3(a).

Case 2: If $h = k$, then at step h, there must exist a nonzero element a_{tm} that leads to the creation of a_{km}. Let $g = T(a_{tm})$. Then it must be true that $g < h$. Also the row t must belong to $\mathcal{P}_h$, i.e., $a_{th} \neq 0$. We discuss two sub-cases.

- **Sub-case 2.1** $g > S(i)$. Namely a_{tm} is created after step $S(i)$. Since $a_{th} \neq 0$, a_{tg} must be nonzero at step g due to the L supernode partitioning. Thus a_{gm} will be nonzero due to the existence of a_{tg} and a_{tm}. Notice that $g < h = k$. This is a contradiction. This situation is depicted in Figure 3(b).
- **Sub-case 2.2** $g \leq S(i)$. We know that a_{th} is nonzero, we discuss two situations according to L supernode partitioning.

  If $t \geq S(i+1)$, the subrow t in the supernode column block i must be dense after the L supernode partitioning. Then $a_{t,S(i)}$ must be nonzero at step $S(i)$. This leads to the creation of $a_{S(i),m}$, which is a contradiction. This is illustrated in Figure 3(c).

  If $t < S(i+1)$, the lower triangular part of L_{ii} is dense after the L supernode partitioning and $a_{t,S(i)}$ must be nonzero at step $S(i)$ by Fact 2. This also leads to a contradiction.

■

The above theorem shows that U partitioning can identify a rich set of dense subcolumns or even dense submatrices in a U factor. We also further incorporate this result with supernode amalgamation in Section 7 and our experiments indicate that more than 64% of numerical updates is performed by the BLAS-3 routine `DGEMM` in S^*, which shows the effectiveness of the U partitioning method. Figure 4 shows the result of a supernode partitioning on a 7×7 sample sparse matrix. One can see that all the submatrices in the upper triangular part of the matrix only contain dense subcolumns.

Using the above theorem, we can further show the structural relationship between two submatrices in the same supernode column block, which will be useful in implementing our algorithm to detect nonzero structures efficiently for numerical updating.

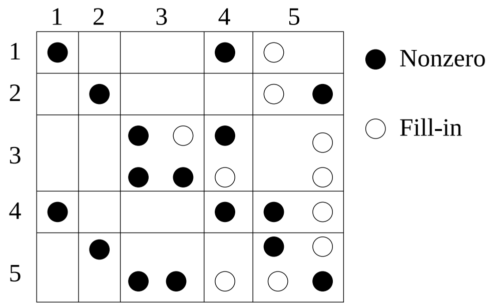

Figure 4: An example of L/U supernode partitioning.

Corollary 1 *Given two nonzero submatrices U_{ij}, $U_{i'j}$ where $i < i' < j$ and $L_{i'i}$ is non-zero. If subcolumn k in U_{ij} is dense, then subcolumn k in $U_{i'j}$ is also dense.*

Proof: The corollary is illustrated in Figure 5. Since $L_{i'i}$ is nonzero, there must be a dense subrow in $L_{i'i}$. This will lead to a nonzero element in the subcolumn k in $U_{i'j}$ because the subcolumn k of U_{ij} is dense. According to Theorem 1, subcolumn k in $U_{i'j}$ is dense. ∎

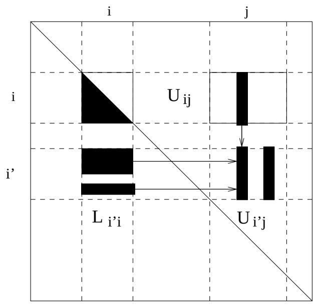

Figure 5: An illustration for Corollary 1.

Corollary 2 *Given two nonzero submatrices U_{ij}, $U_{i'j}$ where $i < i' < j$ and $L_{i',i}$ is nonzero. If U_{ij} is dense, $U_{i'j}$ must be dense.*

Proof: That is straightforward using Corollary 1. ∎

5 Program partitioning and task dependence graphs

After dividing a sparse matrix A into submatrices using the L/U supernode partitioning, we need to partition the LU code accordingly to define coarse grained tasks which can effectively manipulate the partitioned dense data structures. For the convenience of illustration, we use a two-dimensional $N \times N$ array $\mathcal{B}$ to represent the submatrix structure of A, where N is the number of submatrices in both dimensions. That $\mathcal{B}(i, j)$ is a nonzero implies that the corresponding submatrix L_{ij} or U_{ij} in A is nonzero. Therefore A is viewed as a 2-D block structure.

The granularity of the program partitioning is directly related to the available parallelism. A 2-D submatrix oriented partitioning is the desirable granularity because it gives maximum parallelism. However it may not be efficient for parallel sparse partial pivoting on distributed memory machines. If the submatrices in the same column block are assigned to different processors, it will require a substantial number of inter-processor messages to conduct pivot searching and row interchanges. In order to reduce communication, we keep the submatrices, both from L and U part, of the same column block in the same processor. This strategy has the advantage that both pivot searching and row interchange can be done locally.

The column block partitioning follows the supernode structure on the sparse matrix. Each column block k is associated with two types of tasks: $Factor(k)$ and $Update(i,k)$ for some $i < k$. Task $Factor(k)$ is to factorize all the columns in the k-th column block, including finding the pivoting sequence associated with those columns. Instead of performing the row interchange to the other part of the matrix right after each pivoting search, a technique called "delayed-pivoting" is used [3]. In this technique, the pivoting sequence is held until the factorization of the k-th column block is completed. Then the pivoting sequence is applied to the rest of the matrix, i.e., interchange rows. Delayed-pivoting is important, especially to the parallel algorithm, because it is equivalent to aggregating multiple small messages into a larger one. We will vectorize messages as much as possible to reduce expensive communication start-up overhead. Task $Update(i,k)$ exists only for those i's such that $\mathcal{B}(i,k)$ is not zero. It receives column block i together with the pivoting information to update column block k. Here again the pivoting sequence is further packed together with the content of the i-th column block.

An outline of the partitioned sparse LU factorization algorithm with partial pivoting is described in Figure 6. The code of $Factor(k)$ is summarized in Figure 7. It uses BLAS-1 and BLAS-2 subroutines. The computational cost of the numerical factorization is mainly dominated by $Update()$ tasks. The implementation of task $Update(k,j)$ is presented in Figure 8. The lines `(04) and (11)` are using dense matrix multiplications.

```
for k = 1 to N
      Perform task Factor(k);
      for j = k+1 to N with B(k,j) ≠ 0
            Perform task Update(k,j);
      endfor
endfor
```

Figure 6: A partitioned sparse LU factorization with partial pivoting.

We use DAGs to model the irregular parallelism arising in this partitioned sparse LU program. Previous work on sparse Cholesky factorization has used elimination trees (e.g. [21]), which can accurately capture the available parallelism because the input matrix is symmetric and pivoting is not involved. For nonsymmetric sparse LU, one can also induce some parallelism from a row-wise elimination tree [12], but it is not as straightforward as the one for sparse Cholesky. Dynamically created DAGs have been used for modeling parallelism in a nonsymmetric multi-frontal method [17].

```
Factor(k)
    for m = S(k) to S(k+1) - 1
        Find the pivoting position t in column m;
        Interchange row t and row m of the column block k;
        Scale column m and update rest of columns in this column block;
    endfor
```

Figure 7: The description of task $Factor(k)$.

```
Update(k, j)
    Interchange rows according to the pivoting sequence;
    Let L_t be the lower triangular part of L_kk;
    if the submatrix U_kj is dense
        U_kj = L_t^{-1} * U_kj;
    else for each dense subcolumn c_u of U_kj
        c_u = L_t^{-1} * c_u;
    endfor

    for each nonzero submatrix B_ij, i > k in column block j
        if the submatrix U_kj is dense
            B_ij = B_ij - L_ik * U_kj;
        else for each dense subcolumn c_u of U_kj
            Let c_b be the corresponding dense subcolumn of B_ij;
            c_b = c_b - L_ik * c_u;
        endfor
    endfor
```

Figure 8: The description of task $Update(k, j)$.

Given the task definitions in Figures 6, 7 and 8 we can define the structure of a sparse LU task graph in the following.

- There are N tasks $Factor(k)$ (or $F(k)$ for short), where $1 \leq k \leq N$.
- There is a task $Update(k, j)$ (or $U(k, j)$ for short) if $\mathcal{B}(k, j) \neq 0$, where $1 \leq k < j \leq N$. For a dense matrix, there will be a total of $N(N-1)/2$ updating tasks.
- There is a dependence edge from $F(k)$ to task $U(k, j)$, for any j such that $\mathcal{B}(k, j) \neq 0$.
- There is a dependence from $U(i, k)$ to $U(i', k)$, where $i < i'$ and there exists no task $U(t, k)$ such that $i < t < i'$.

Figure 9(a) shows the nonzero pattern of the partitioned matrix shown in Figure 4. Figure 9(b) is the corresponding task dependence graph.

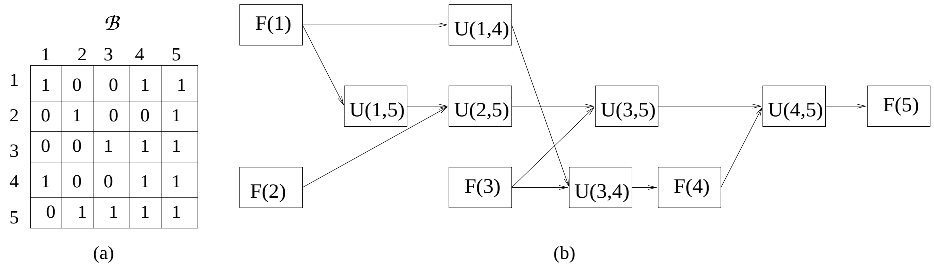

Figure 9: (a) The structure information array $\mathcal{B}$ for the example in Figure 4. (b) The dependence graph derived from the partitioning result.

6 Task scheduling and run-time support

We discuss how sparse LU tasks can be mapped onto processors so that parallel time can be minimized and how a schedule can be executed efficiently.

6.1 Scheduling sparse LU tasks on distributed memory machines

For dense problems, one can use a regular mapping such as block cyclic mapping [15]. These schemes usually result in good load balancing and decent speedup. However it is not agreed that what the best mapping strategy is for irregular applications such as sparse matrix computations.

For sparse Cholesky factorization, researchers have used heuristic mapping strategies [19, 22] based on elimination trees. For sparse LU, an elimination tree does not directly reflect the available parallelism and therefore scheduling based on the elimination tree is difficult. [12] uses a dynamic load balancing algorithm on a shared memory machine. For distributed memory machines, dynamic and adaptive load balancing work well for problems with very coarse grained computations, but it is still an open problem to balance the benefits of dynamic scheduling with the run-time control overhead since task and data migration cost is too expensive for sparse problems with mixed granularities.

We use task dependence graphs to guide scheduling and have investigated two types of scheduling schemes. In both schemes, the owner-compute rule is used to assign all tasks that update the same column block to the same cluster so that unnecessary communication can be eliminated.

- **Compute-ahead scheduling**. This is to use block-cyclic mapping of tasks with a compute-ahead execution strategy, which is demonstrated in Figure 10. This idea has been used to speed up parallel dense factorizations [15]. It executes the numerical factorization layer by layer based on the current submatrix index. The parallelism is exploited for concurrent updating. In order to overlap computation with communication, the factorization $F(k+1)$ is executed as soon as $F(k)$ and $U(k, k+1)$(if exists) finish so that the pivoting sequence and column block $k+1$ for the next layer can be broadcast as early as possible.
- **Graph scheduling**. We assign clusters to processors and order task execution within each

processor using the algorithms in [24]. The basic optimizations are balancing processor loads and overlapping computation with communication to hide communication latency. These are done by utilizing global dependence structures and critical path information.

```
if column block 1 is local
     Perform task Factor(1);
     Broadcast column block 1 and the pivoting sequence;
for k = 1 to N − 1
     if column block k + 1 is local
          if B(k, k + 1) ≠ 0
               Receive column block k and the pivoting choices;
               Interchange rows according to the pivoting sequence;
               Perform task Update(k, k + 1);
          Perform task Factor(k + 1);
          Broadcast column block k + 1 and the pivoting sequence;
     for j = k + 2 to N with B(k, j) ≠ 0
          if column block j is local
               if column block k has not been received
                    Receive column block k and the pivoting choices;
               Interchange rows according to the pivoting sequence;
               Perform task Update(k, j);
     endfor
endfor
```

Figure 10: The parallel sparse LU factorization algorithm with partial pivoting using compute-ahead schedule.

Graph scheduling has already been shown effective for irregular problems such as adaptive n-body simulation [13] and sparse Cholesky factorization [8, 9]. For sparse LU, it can better exploit irregular parallelism compared to the compute-ahead scheduling. The major reason is that the compute-ahead scheduling is not as efficient in overlapping communication with computation. We demonstrate this point using the LU task graph in Figure 9. For this example, the Gantt charts of the compute-ahead schedule and the schedule derived by our graph scheduling algorithm are listed in Figure 11. It is assumed that each task has a computation weight 2 and each edge has communication weight 1. It is easy to see that our scheduling approach produces a better result than the compute-ahead schedule. If we look at the compute-ahead schedule carefully, we can see that the reason is that it can look ahead only one step so that the execution of the task $F(3)$ is placed after $U(1,5)$. On the other hand, the graph scheduling algorithm detects that $F(3)$ can be executed before $U(1,5)$ which leads to better overlap of communication with computation. Also the simple cyclic mapping could result in load imbalance, and the graph scheduling algorithm can do a better job in balancing processor loads.

In conclusion, our graph scheduling scheme outperforms the compute-ahead scheduling in a theoretical sense. However, the efficient execution of a task graph schedule is the key to reach this goal. We discuss the run-time support strategies for executing tasks in the next subsection.

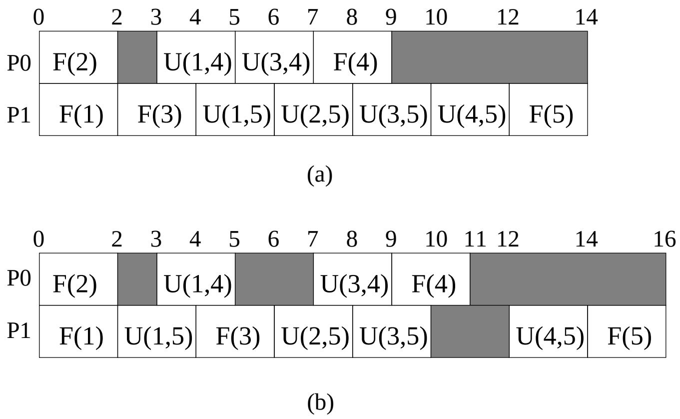

Figure 11: (a) A schedule derived by our graph scheduling algorithm. (b) A compute-ahead schedule.

6.2 The RAPID run-time system

We have implemented our scheme using the RAPID run-time system [9]. This system provides a set of library functions for specifying irregular data objects and tasks that access these objects. The system then extracts a task dependence graph from data access patterns, schedules and executes tasks efficiently on a distributed memory machine.

The RAPID run-time support provides an efficient task communication protocol to execute scheduled tasks. We summarize the main optimizations as follows.

1. Eliminating the buffering and copying overhead in the asynchronous communication. The key idea is to use Remote Memory Access(RMA) to communicate a data object between two processors. It does not incur any copying/buffering during a data transfer. The functionality of RMA suffices to fit our task protocol and the overhead is kept as small as possible. RMA is available in modern multi-processor architectures such as Cray-T3D [23] and Meiko CS-2 [8].

2. Eliminating redundant communication. A task may send the same message to several successors and some of these successors could be assigned to the same processor. In this case, it is enough to send this message once to the processor on which those successors reside [24].

3. Eliminating unnecessary synchronizations. Since the RMA directly writes data to a remote address, it is possible that the content at the remote address is still being used by other tasks and then the execution at the remote processor could be incorrect. Thus for a general computation, a permission to write the remote address needs to be obtained before issuing a remote write. However in the RAPID system, this hand-shaking process can be avoided by a carefully designed task communication protocol [9].

It should be mentioned that un-synchronized direct remote memory access may overwrite some live data and cause data inconsistencies during the execution. In [8, 9], we prove that if a task graph is *dependence-complete*, i.e., all data dependencies are captured in this graph and the graph is sequentializable, the task schedule execution by RAPID is always correct. It is easy to show that our sparse LU task graphs are dependence-complete. Hence there is no data inconsistency during the schedule execution.

As it will be seen in Section 8.2, the above run-time optimization strategies are effective in executing sparse LU graphs. The experiments show that more than 70% of the speedup predicted by the scheduler can be delivered on Cray-T3D. In addition, using RAPID system greatly reduces the amount of work to parallelize the sparse LU algorithm.

7 Supernode amalgamation

For most tested sparse matrices, the average size of a supernode after L/U partitioning is very small, about 1.5 to 2 columns. This results in very fine grained tasks. Amalgamating small supernodes can lead to great performance improvement for both parallel and sequential sparse codes because it can improve cache performance and reduce inter-processor communication overhead.

There could be many ways to amalgamate supernodes [4, 21]. The basic idea is to relax the restriction that all the columns in a supernode must have exactly the same nonzero structure. The amalgamation is usually guided by a supernode elimination tree. A parent could be merged with its children if the merge does not create too many extra zero entries. Row and column permutations are needed if the parent is not consecutive with its children.

However, the column and/or row permutation introduced by the above amalgamation method could undermine the correctness of the static symbolic factorization. We give an example to demonstrate this point. The left part of Figure 12 shows a sparse structure after the static symbolic factorization. Assume that column 1 and row 2 are interchanged by a permutation. The right part of Figure 12 depicts the result of the interchange. Then the new structure does not correctly predict nonzero patterns for all possible pivoting sequences. Suppose that at step 1 of the numerical factorization, the fourth row is chosen as the pivot row. Then there will be no space in the first row to accommodate the two nonzero elements swapped from the fourth row (indicated by the question marks in Figure 12). Similarly we can find counter examples showing that row permutation or column/row permutation can cause the result of static symbolic factorization incorrect.

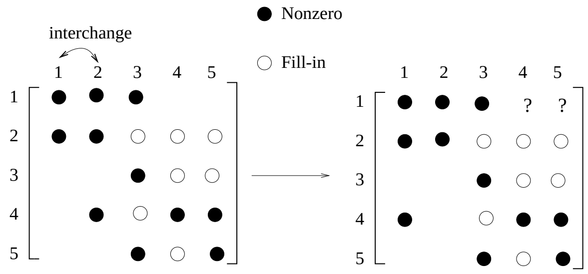

Figure 12: An example showing how a column permutation could invalidate the result of symbolic factorization.

We have used a simple approach that does not require any permutation. This approach only amalgamates consecutive supernodes if their nonzero structures only differ by a small number of entries. We can control the maximum allowed differences by an amalgamation factor r. Our experiments show that when r is in the range of $4-6$, it gives the best performance for the tested matrices and leads to $16-50\%$ improvement on the execution times of the sequential code. Compared to the $5-15\%$ running time improvement by relaxed supernodes in [4], our improvement is much more. The reason is that by getting bigger supernodes, we are getting larger dense structures, although there may be a few zero entries in them, and we are taking more advantage of BLAS-3 kernels. Notice that after applying the supernode amalgamation, the dense structures identified in the Theorem 1 are not strictly dense any more. We call them *almost-dense* structures and can still use the result of Theorem 1 with a minor revision. That is summarized in the following corollary. All the results presented in Section 8 are obtained using supernode amalgamation.

Corollary 3 *Given a sparse matrix A, if supernode amalgamation is applied to A after the static symbolic factorization and 2-D nonsymmetric L/U supernode partitioning are performed on A, each nonzero submatrix in the U factor of A contains only almost-dense subcolumns.*

8 Experimental studies

Our experiments are conducted on a Cray-T3D distributed memory machine. Each processing element of T3D includes a DEC Alpha EV4(21064) processor with 64 megabytes of memory. The cache size is 8K per processor. The BLAS-3 matrix-matrix multiplication routine `DGEMM` can achieve 103 MFLOPS, and the BLAS-2 matrix-vector multiplication routine `DGEMV` can reach 85 MFLOPS. These figures are obtained assuming all the data is in cache and using cache read-ahead optimization on T3D. The communication network of the T3D is a 3-D torus. The Cray provides a shared memory access library called `shmem` which can achieve 126 MBytes/s bandwidth and $2.7\mu s$ communication overhead using `shmem_put()` primitive [23]. The implementation of the RAPID system on T3D uses `shmem_put()` for remote memory access.

8.1 Sequential performance

The sequential performance on the testing matrices from Table 1 are listed in Table 3. In order to evaluate our sequential code, we also list the performance of SuperLU on these matrices. In calculating the achieved MFLOPS, we use the operation count obtained from SuperLU and do not include extra floating point operations introduced by the overestimation.

Though the static symbolic factorization has introduced a lot of extra computation work, the performance of S^* after nonsymmetric L/U partitioning is consistently competitive to that of highly optimized SuperLU. The absolute single node performance that has been achieved by the S^* approach is in the range of $6-10\%$ of the `DGEMM` performance for those matrices of small or medium sizes. Considering the fact that sparse codes usually suffer poor cache reuse, this performance is

Matrix	S^* Approach		SuperLU		Exec Time Ratio
	Seconds	MFLOPS	Seconds	MFLOPS	S^*/SuperLU
1	2.87	8.81	2.39	10.57	1.21
2	6.12	6.90	4.94	8.56	1.27
3	6.67	6.71	5.30	8.45	1.26
4	6.06	10.18	4.27	14.46	1.56
5	2.11	8.24	1.62	10.66	1.34
6	6.03	10.37	4.09	15.25	1.66
7	10.38	10.31	7.18	14.95	1.60

Table 3: Sequential performance on T3D: S^* versus SuperLU.

reasonable. In addition, the amount of computation for these testing matrices is small, ranging from 18 to 107 million double precision floating operations. Since the characteristic of the S^* approach is to explore more dense structures and utilize BLAS-3 kernels, better performance is expected on larger or denser matrices. This is verified empirically in Table 4. The first matrix `goodwin` is a relatively large sparse matrix and S^* reaches 15% of `DGEMM` performance. Matrix `b33_5600` is truncated from `BCSSTK33` because of the current implementation is not able to handle the entire matrix due to memory constraint. For a 1000×1000 dense matrix, 63.6 MFLOPS has been achieved, which is about 62% of the peak BLAS-3 performance and 68% of the LAPACK `DGETRF` performance. And for the same dense matrix, SuperLU achieves 34 MFLOPS.

Matrix	Order	#flops($\times 10^6$)	Seconds	MFLOPS
goodwin	7320	665.0	43.72	15.2
b33_5600	5600	2,529.0	110.52	22.9
dense1000	1000	666.2	10.48	63.6

Table 4: Sequential performance of the S^* approach for larger or denser matrices on T3D.

We present a quantitative analysis to explain why S^* can be competitive to SuperLU. We quantify the percentage of numerical updates that is performed in the S^* approach using BLAS-3 kernel in Table 6. Assume the speed of BLAS-2 kernel is ω_2 *second/flop* and the speed of BLAS-3 kernel is ω_3 *second/flop*. The total amount of numerical updates is $\mathcal{C}$ *flops* for SuperLU and $\mathcal{C}'$ *flops* for the S^*. Apparently $\mathcal{C}' \geq \mathcal{C}$. For simplicity, we ignore the computation from scaling part within each column because it contributes very little to the total execution time. Hence we have:

$$T_{SuperLU} = T_{symbolic} + \omega_2 * \mathcal{C}, \quad and \tag{1}$$

$$T_{S^*} = (1-\rho) * \omega_2 * \mathcal{C}' + \rho * \omega_3 * \mathcal{C}', \tag{2}$$

where $T_{symbolic}$ is the time spent on dynamic symbolic factorization in the SuperLU approach, ρ is the percentage of the numerical updates that are performed by `DGEMM` in S^*. Let η be the ratio of symbolic factorization time to numerical factorization time in SuperLU, then we simplify Equation 1 to the following:

$$T_{SuperLU} = (1 + \eta) * \omega_2 * \mathcal{C}. \tag{3}$$

Matrix	1	2	3	4	5	6	7
$\frac{flops(S^*)}{flops(SuperLU)}$	3.50	4.77	4.88	3.56	3.68	3.68	3.83

Table 5: Ratios of the number of floating point operations performed for numerical factorization in S^* and in SuperLU.

We estimate that $\eta \approx 0.82$ for the tested matrices based on the results in [4]. From Table 6 we can see that $\rho \approx 0.67$. We also calculate the ratios of the number of floating point operations performed in S^* and SuperLU for the tested matrices in Table 5. In average, the value of $\frac{\mathcal{C}'}{\mathcal{C}}$ is 3.98. We plug in these typical parameters in Equation 2 and 3, and we have:

$$\frac{T_{S^*}}{T_{SuperLU}} \approx \frac{(0.33 * \omega_2 + 0.67 * \omega_3) * 3.98}{1.82 * \omega_2}. \tag{4}$$

For T3D, $\omega_2 = \frac{1}{85}\ \mu s/flop$ and $\omega_3 = \frac{1}{103}\ \mu s/flop$. Then we can get $\frac{T_{S^*}}{T_{SuperLU}} \approx 1.93$, which is close to the ratios obtained in Table 3. The discrepancy is caused by the fact that the submatrix sizes of supernodes are non-uniform, which leads to different caching performance. If submatrices are of uniform sizes, we expect our prediction is more accurate. For instance, in the dense case, $\frac{\mathcal{C}'}{\mathcal{C}}$ is exactly 1. The ratio $\frac{T_{S^*}}{T_{SuperLU}}$ is calculated as 0.48, which is almost the same as the ratio of the MFLOPS obtained: $34/63.6 = 0.53$.

The above analysis shows that using BLAS-3 as much as possible makes S^* competitive to SuperLU. Suppose in a machine that `DGEMM` outperforms `DGEMV` substantially in terms of cache performance, S^* could be faster than SuperLU. Table 6 also shows that ratios of execution times for S^* with and without using BLAS-3 routines. $T_{mm}(S^*)$ is the execution time obtained by using BLAS-3 routines as much as possible. $T_{mv}(S^*)$ is the execution time by replacing all the BLAS-3 routines in S^* by BLAS-2 routines. As a result, $T_{mm}(S^*)$ is usually 20% less than $T_{mv}(S^*)$. This can again be verified by Equation 4 as follows:

$$\frac{T_{mm}(S^*)}{T_{mv}(S^*)} = \frac{(1 - 0.67) * \omega_2 + 0.67 * \omega_3}{\omega_2} \approx 0.88.$$

8.2 Parallel performance

In this subsection, we report a set of experiments conducted to examine the overall parallel performance of S^* and the effectiveness of its optimization strategies.

Matrix	1	2	3	4	5	6	7
BLAS-3 Percentage	67%	69%	66%	68%	64%	66%	68%
$T_{mm}(S^*)/T_{mv}(S^*)$	0.81	0.81	0.77	0.81	0.83	0.80	0.80

Table 6: Percentage of the numerical updates which is performed using BLAS-3 routines and the ratios of the execution times with and without using BLAS-3 routines.

Overall Performance: The overall speedups of S^* for the tested matrices are shown in Figure 13. S^* scales fairly well up to 32 processors for all the matrices. The speedups range from 6.3 to 11.4 on 32 processors. However the speedups remain almost the same on 64 processors for those matrices except `goodwin` and `b33_5600`, which are much larger problems than the rest. The reason is that those small tested matrices do not have enough amount of computation to saturate a large number of processors. The execution times for most of these matrices are less than 1 second on 32 processors. Even for `goodwin` the execution time on 64 processors is about 3 seconds and therefore an absolute performance of 216 MFLOPS has been achieved. For `b33_5600`, 412 MFLOPS has been achieved on 64 processors, and 438 MFLOPS on 128 processors. It is our belief that better and more scalable performance can be obtained on larger matrices. This trend can be seen on the matrix `goodwin` and b33_5600. But currently the available memory on each node of T3D limits the problem size that can be solved with the current version of S^*. We are looking at the issues of how to execute large task graphs under memory constraints.

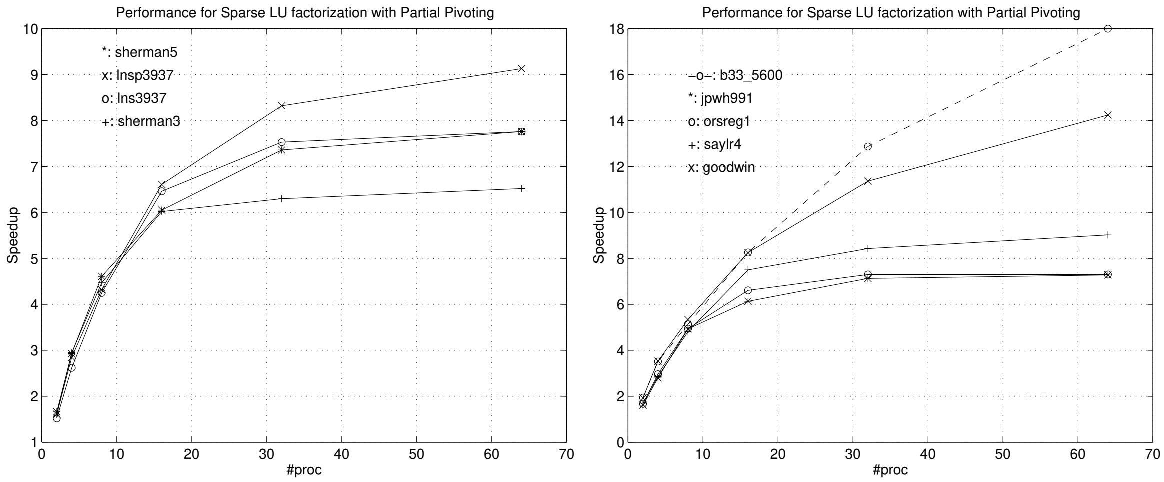

Figure 13: Parallel speedups on Cray-T3D.

Effectiveness of the Graph Scheduling: We have implemented our sparse LU code using a column-block cyclic mapping with compute-ahead schedule (referred to as $S^*(CA)$) and compared the performance with that of the code (referred to as $S^*(RAPID)$) which uses the RAPID system to schedule and execute tasks. Note that in the implementation of executing a compute-ahead schedule, the communication optimizations used in the RAPID system are still included. The results are shown in Figure 14. The Y axis is $1 - PT_{S^*(RAPID)}/PT_{S^*(CA)}$, where PT stands for

Parallel Time. For 2 and 4 processors, in certain cases, the compute-ahead code is slightly faster than the RAPID code. But for the number of processors more than 4, the RAPID code runs $25-68\%$ faster. The more processors involved, the bigger the performance gap is. In fact, from 16 processors to 32 processors, in all cases the parallel times of the compute-ahead code actually increase. The reason is that for a small number of processors, there are sufficient tasks making all processors busy and the compute-ahead schedule performs well while the RAPID code suffers a certain degree of control overhead. For a larger number of processors, schedule optimization becomes important since there is limited parallelism to exploit.

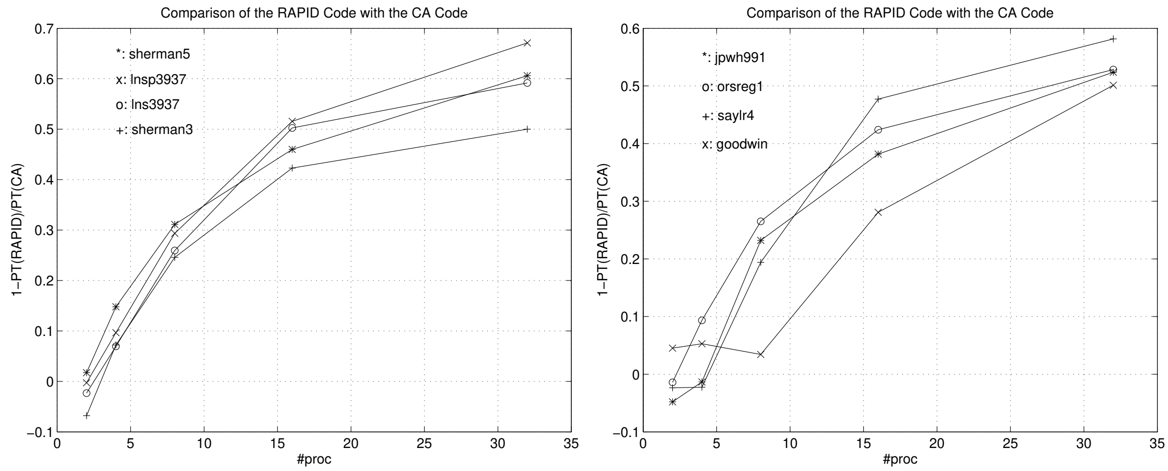

Figure 14: Impact of different scheduling and mapping schemes on Cray-T3D.

We have also run both codes on a dense matrix of size 1000×1000, and we list the results in Table 7. Not surprisingly, the compute-ahead code outperforms the RAPID code in most cases because there is sufficient parallelism to exploit. But in the 32 processor case, the RAPID code is still faster than the compute-ahead code. It is safe to conclude that the graph scheduling algorithm is effective in mapping irregular problems.

#proc	2	4	8	16	32
$\frac{S^*(CA)}{S^*(RAPID)}$	0.96	0.95	0.93	0.89	1.02

Table 7: Ratios of parallel times by RAPID code and compute-ahead code for a dense matrix.

Effectiveness of the Run-time Support: A good scheduling algorithm does not suffice to produce an efficient code. A run-time system is necessary to convert the scheduled performance to real performance. The run-time support of the RAPID system incorporates several communication optimizations and provides a very low overhead task communication protocol. We plot the difference between the actual speedups on T3D and the speedups predicted by the scheduler in Figure 15. The Y axis is the ratio $1 - Speedup_{T3D}/Speedup_{Scheduler}$. The parallel code using RAPID has been able to achieve more than 70% of the scheduled speedup. In certain cases, the actual speedups are more than the predicted speedups. This result is a clear indication that the RAPID system is suitable for parallelizing the sparse pivoting code. The run-time overhead is small enough to make the run-time situation match the static schedule closely.

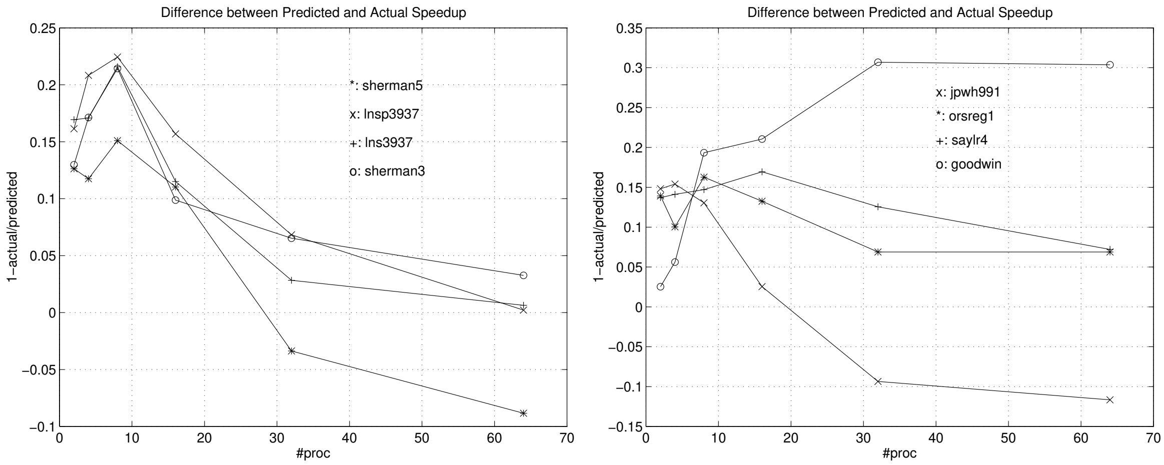

Figure 15: Difference between actual and predicted speedups on Cray-T3D.

Effectiveness of supernode amalgamation: We have examined how effective our supernode amalgamation strategy is. Let PT_a and PT be the parallel time with and without supernode amalgamation respectively. The parallel time improvement ratio $1 - PT_a/PT$ for several testing matrices are listed in Table 8. Apparently the supernode amalgamation has brought significant improvement due to the increase of supernode size which implies an increase of the task granularities. This is important to obtain good parallel performance [14].

Matrix	P=1	P=2	P=4	P=8	P=16	P=32
1	48%	53%	56%	56%	59%	47%
4	20%	31%	34%	41%	27%	20%
5	50%	55%	56%	61%	56%	48%
6	16%	23%	31%	27%	22%	14%
7	20%	26%	27%	32%	36%	25%

Table 8: Parallel time improvement obtained by supernode amalgamation.

9 Conclusions

In this paper we present a new approach for efficiently parallelizing sparse LU factorization with partial pivoting on distributed memory machines. The major contribution of this paper is that we integrate several optimization techniques together such as static symbolic factorization, graph scheduling and run-time support for asynchronous parallelism, extend nonsymmetric supernode partitioning techniques to effectively identify dense structures, and maximize the use of BLAS-3 subroutines in the algorithm design. Using these ideas, we are able to exploit more data regularity for this open irregular problem and achieve decent performance on both uni-processor and multi-processors. Our current experiments are conducted on matrices of small or medium sizes, we believe this approach has a great potential to get better performance on larger matrices and plan to investigate this in future work. It should be noted that the static symbolic factorization could fail to be practical if the input matrix has a nearly dense row because it will lead to an almost complete fill-in of the whole matrix. Fortunately this is not the case in most of matrices derived from real applications. Therefore our approach is still applicable to a wide range of problems.

Acknowledgment

This work is supported by NSF CCR-9409695 and a startup fund from UCSB. We would like to thank Xiaoye Li and Jim Demmel for helpful discussion and providing us their testing matrices and SuperLU code, Dan Andresen, Apostolos Gerasoulis, Ed Rothberg, Rob Schreiber, Horst Simon and Kathy Yelick for their valuable comments, and Stefan Boeriu for his help in accessing Cray-T3D.

References

[1] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon. Progress in Sparse Matrix Methods for Large Sparse Linear Systems on Vector Supercomputers. *International Journal of Supercomputer Applications*, 1:10–30, 1987.

[2] T. Davis. User's guide for the Unsymmetric-pattern Multifrontal Package (UMFPACK). Technical Report TR-93-020, Computer and Information Sciences Department, University of Florida, June 1993.

[3] J. Demmel. Numerical Linear Algebra on Parallel Processors. Lecture Notes for NSF-CBMS Regional Conference in the Mathematical Sciences, June 1995. To be published as a book by SIAM.

[4] J. Demmel, S. Eisenstat, J. Gilbert, X. Li, and J. Liu. A Supernodal Approach to Sparse Partial Pivoting. Technical Report CSD-95-883, UC Berkeley, September 1995.

[5] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An Extended Set of Basic Linear Algebra Subroutines. *ACM Trans. on Mathematical Software*, 14:18–32, 1988.

[6] I. S. Duff. On Algorithms for Obtaining a Maximum Transversal. *ACM Transactions on Mathematical Software*, 7(3):315–330, September 1981.

[7] I. S. Duff. Personal Communication, 1996.

[8] C. Fu and T. Yang. Efficient Run-time Support for Irregular Task Computations with Mixed Granularities. In *Proceedings of IEEE International Parallel Processing Symposium*, pages 823–830, Hawaii, April 1996.

[9] C. Fu and T. Yang. Run-time Compilation for Parallel Sparse Matrix Computations. In *Proceedings of ACM International Conference on Supercomputing*, pages 237–244, Philadelphia, May 1996.

[10] K. Gallivan, B. Marsolf, and H. Wijshoff. The Parallel Solution of Nonsymmetric Sparse Linear Systems using H* Reordering and an Assoicated Factorization. In *Proc. of ACM International Conference on Supercomputing*, pages 419–430, Manchester, July 1994.

[11] A. George and E. Ng. Symbolic Factorization for Sparse Gaussian Elimination with Partial Pivoting. *SIAM J. Scientific and Statistical Computing*, 8(6):877–898, November 1987.

[12] A. George and E. Ng. Parallel Sparse Gaussian Elimination with Partial Pivoting. *Annals of Operations Research*, 22:219–240, 1990.

[13] A. Gerasoulis, J. Jiao, and T. Yang. Scheduling of Structured and Unstructured Computation . In D. Hsu, A. Rosenberg, and D. Sotteau, editors, *Interconnections Networks and Mappings and Scheduling Parallel Computation.* American Math. Society, 1995.

[14] A. Gerasoulis and T. Yang. On the Granularity and Clustering of Directed Acyclic Task Graphs . *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, June 1993.

[15] G. Golub and J. M. Ortega. *Scientific Computing: An Introduction with Parallel Computing Compilers* . Academic Press, 1993.

[16] A. Gupta and V. Kumar. Optimally Scalable Parallel Sparse Cholesky Factorization. In *Proc. of 7th SIAM Conf. on Parallel Processing for Scientific Computing*, pages 442–447, February 1995.

[17] S. Hadfield and T. Davis. A Parallel Unsymmetric-pattern Multifrontal Method. Technical Report TR-94-028, Computer and Information Sciences Departmenmt, University of Florida, August 1994.

[18] M. Heath, E. Ng, and B. Peyton. Parallel Algorithms for Sparse Linear Systems . *SIAM Review*, 33(3):420–460, September 1991.

[19] J. W. H. Liu. Computational Models and Task Scheduling for Parallel Sparse Cholesky Factorization . *Parallel Computing*, 18:327–342, 1986.

[20] R. Lucas, T. Blank, and J. Tiemann. A Parallel Solution Method for Large Sparse Systems of Equations. *IEEE Trans. on Computer-Aided Design*, CAD-6(6):981–991, November 1987.

[21] E. Rothberg. *Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization* . PhD thesis, Dept. of Computer Science, Stanford, December 1992.

[22] E. Rothberg and R. Schreiber. Improved Load Distribution in Parallel Sparse Cholesky Factorization. In *Proc. of Supercomputing'94*, pages 783–792, November 1994.

[23] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling Synchronization and Data Transfer in Message Passing Systems of Parallel Computers. In *Proc. of International Conference on Supercomputing*, pages 1–10, Barcelona, July 1995.

[24] T. Yang and A. Gerasoulis. PYRROS: Static Task Scheduling and Code Generation for Message-Passing Multiprocessors . In *Proc. of 6th ACM International Conference on Supercomputing*, pages 428–437, 1992.